



JavaScript Arrays

(Detailed notes yet beginner friendly)

What is an Array (in general) ?

- An array is a numbered, ordered list of items.
- stores a collection of elements (items) of the same data type in contiguous (adjacent) memory locations

Key Characteristics of Arrays

Contiguous Memory : Elements are stored next to each other, improving cache friendliness.

Indexed Access : Each element is accessed by a numeric index (e.g., `arr[0]`).

Fixed Size : Traditionally, arrays have a predetermined size that cannot be changed after creation (though some modern languages allow dynamic resizing).

Homogeneous Elements : Typically, all elements in an array must be of the same data type (e.g., all integers or all strings), though some languages allow variations, like JavaScript and Python.

Creating an array

We can create array using simple square brackets [].

```
// An empty array (an empty list)
let emptyList = [];

// An array of numbers
let scores = [98, 85, 100, 92];

// An array of strings
let names = ["Alice", "Bob", "Charlie"];

// An array can hold different data types
let mixedData = [10, "hello", true, null];
```

The .length Property

.length property is used to check length of an array, i.e. how many items are there in the array.

```
let fruits = ["Apple", "Banana", "Cherry"];

console.log(fruits.length); // 3
```

NOTE: This property automatically updated when we add/remove items.

Accessing and Changing Elements (Zero-Based Indexing)

To read or change an item, you use its position number, called an **index**.

CRITICAL RULE: **Array indexing in JavaScript starts at 0.**

- The 1st item is at index 0.
- The 2nd item is at index 1.
- ...and so on.

```
let fruits = ["Apple", "Banana", "Cherry"];  
// Index:    0          1          2  
  
// Accessing (reading) an element  
console.log(fruits[0]); // "Apple"  
  
// Changing (writing) an element  
fruits[1] = "Blueberry";  
console.log(fruits); // ["Apple", "Blueberry", "Cherry"]  
  
// Getting the last element  
console.log(fruits[fruits.length - 1]); // "Cherry"
```

NOTE: If you try to access an index that doesn't exist, you get **undefined**

Arrays in JavaScript, is mutable by default, meaning their applying basic modification methods will change original array.

A. Adding/Removing from the END of the Array

- **.push(item1, item2, ...)** : Adds one or more items to the end.
- **.pop()** : Removes the last item and gives it back to you.

```
let tasks = ["Wash dishes"];
tasks.push("Do laundry", "Buy groceries");
console.log(tasks); // ["Wash dishes", "Do laundry", "Buy groceries"]
```

```
let completedTask = tasks.pop();
console.log(completedTask); // "Buy groceries"
console.log(tasks); // ["Wash dishes", "Do laundry"]
```

B. Adding/Removing from the BEGINNING of the Array

- **.unshift(item1, item2, ...)**: Adds one or more items to the beginning.
- **.shift()**: Removes the first item and gives it back to you.

```
let queue = ["Person B", "Person C"];
queue.unshift("Person A");
console.log(queue); // ["Person A", "Person B", "Person C"]
```

```
let firstInLine = queue.shift();
console.log(firstInLine); // "Person A"
console.log(queue); // ["Person B", "Person C"]
```

Note: unshift and shift can be slower on very large arrays because every other element needs to be shifted to a new position.

To perform an action on every item in the list, you use a loop.

A. The Classical for loop

This is the most fundamental way to iterate. You create a counter variable (i) that tracks the index.

```
let scores = [98, 85, 100];
let total = 0;

// The loop runs as long as `i` is less than the array's length.
for (let i = 0; i < scores.length; i++) {
  console.log(`Processing score at index ${i}: ${scores[i]}`);
  total = total + scores[i];
}

console.log(`The total score is: ${total}`); // 283
```

B. The for... of loop (Modern and Simple)

This is the preferred method when you only care about the value of each item and not its index. It's cleaner and less error-prone.

```
let names = ["Alice", "Bob", "Charlie"];

for (const name of names) {
  console.log(`Hello, ${name}!`);
}
```

A. The `splice()` Method → (The "Surgery" Tool)

This is a powerful mutating method that can add, remove, or replace elements anywhere in the array.

Syntax: `array.splice(startIndex, deleteCount, item1, item2, ...)`

```
let months = ["Jan", "March", "April", "June"];

// Example 1: REMOVING "March"
// Start at index 1, delete 1 element.
months.splice(1, 1);
console.log(months); // ["Jan", "April", "June"]

// Example 2: ADDING "Feb"
// Start at index 1, delete 0 elements, and add "Feb".
months.splice(1, 0, "Feb");
console.log(months); // ["Jan", "Feb", "April", "June"]

// Example 3: REPLACING "April" with "May"
// Start at index 2, delete 1 element, and add "May".
months.splice(2, 1, "May");
console.log(months); // ["Jan", "Feb", "May", "June"]
```

B. The `slice()` Method (Making a Copy)

This method creates a new array by copying a portion of an existing array. It does not change the original.

Syntax: `array.slice(startIndex, endIndex)`
(end index is not included)

```
let animals = ["ant", "bison", "camel", "duck", "elephant"];

// Copy the elements from index 2 up to (but not including) index 4
let middleAnimals = animals.slice(2, 4);
console.log(middleAnimals); // ["camel", "duck"]

// If you omit the end index, it copies to the end of the array.
let allButFirstTwo = animals.slice(2);
console.log(allButFirstTwo); // ["camel", "duck", "elephant"]

// A common way to make a full copy of an array:
let fullCopy = animals.slice();

console.log(animals); // The original is unchanged!
```


C. The Spread Operator (...) (The Modern Way to Copy/Combine)

This is the most popular way in modern JavaScript to create copies or merge arrays.

```
const arr1 = ["a", "b"];
const arr2 = ["c", "d"];

// Make a copy
const copyOfArr1 = [...arr1];
console.log(copyOfArr1); // ["a", "b"]

// Combine two arrays
const combined = [...arr1, ...arr2];
console.log(combined); // ["a", "b", "c", "d"]

// Add elements in the middle
const withMiddle = [...arr1, "x", "y", ...arr2];
console.log(withMiddle); // ["a", "b", "x", "y", "c", "d"]
```

A. Array to String Conversion

- **.join(separator)** : Joins all elements of an array into a single string. You can specify what to put between them.

```
const names = ["Alice", "Bob", "Charlie"];
const nameList = names.join(", ");
console.log(nameList); // "Alice, Bob, Charlie"
```

B. Simple Searching

- **.indexOf(item)** : Returns the index of the first time an item appears in the array. Returns -1 if not found.
- **.lastIndexOf(item)** : Returns the index of the last time an item appears.
- **.includes(item)** : Returns true or false if an item exists. This is often the simplest and most readable choice for a simple check.

```
const numbers = [10, 20, 30, 20, 40];

console.log(numbers.indexOf(20)); // 1 (the first
occurrence)
console.log(numbers.lastIndexOf(20)); // 3 (the last
occurrence)
console.log(numbers.indexOf(99)); // -1

console.log(numbers.includes(30)); // true
console.log(numbers.includes(99)); // false
```

1. Sorting Arrays: The .sort() Method

The .sort() method sorts the elements of an array in place meaning it mutates (changes) the original array.

NOTE: .sort() has a default behavior that is often not what you want.

A. The Default Sort (The "Dictionary" Sort) By default, with no arguments, .sort() converts all elements to strings and sorts them in lexicographical (dictionary) order based on their UTF-16 character codes.

This works fine for strings:

```
let fruits = ["Cherry", "Apple", "Banana"];
fruits.sort();

console.log(fruits); // ["Apple", "Banana", "Cherry"] (This works as expected)
```

ALERT: But this default behavior is a disaster for numbers.

```
let numbers = [100, 2, 5, 25, 1];
numbers.sort();

// The numbers are converted to strings: "100", "2", "5", "25", "1"
// The sort then compares them character by character:
// "1" comes before "100"
// "100" comes before "2"
// "2" comes before "25"

console.log(numbers); // [1, 100, 2, 25, 5] (This is WRONG!)
```

Because "100" starts with a "1", the dictionary sort places it before "2"

B. The Correct Way to Sort: The Compare Function

To sort anything other than simple strings, you must provide a compare function as an argument to **.sort()**.

The compare function takes two arguments, a and b, which represent two elements from the array being compared. It must return a number with a specific meaning:

- **If it returns a negative number (< 0)** : a should come before b.
- **If it returns a positive number (> 0)** : a should come after b.
- **If it returns 0** : a and b are considered equal, and their order doesn't change relative to each other.

C. Sorting Numbers Correctly

This is where the formula $(a, b) \Rightarrow a - b$ comes in. It's a brilliant and concise way to implement the compare function for numerical sorting.

Let's analyze $(a, b) \Rightarrow a - b$ for ascending order (smallest to largest):

- **Case 1** : Let's say a is 5 and b is 10.
 - $a - b$ is $5 - 10 = -5$ (a negative number).
 - The rule says: if negative, a comes before b . This is correct.
- **Case 2** : Let's say a is 25 and b is 2.
 - $a - b$ is $25 - 2 = 23$ (a positive number).
 - The rule says: if positive, a comes after b . This is correct.
- **Case 3** : Let's say a is 100 and b is 100.
 - $a - b$ is $100 - 100 = 0$.
 - The rule says: if zero, the order doesn't matter. This is correct.

```
let numbers = [100, 2, 5, 25, 1];
```

```
// Sort in ascending order
```

```
numbers.sort((a, b) => a - b);
```

```
console.log(numbers); // [1, 2, 5, 25, 100] (Correct!)
```

```
// To sort in descending order, just flip the subtraction.
```

```
numbers.sort((a, b) => b - a);
```

```
console.log(numbers); // [100, 25, 5, 2, 1] (Correct!)
```

2. Flattening Arrays: The `.flat()` Method (ES2019)

An array can contain other arrays, creating a nested or "multidimensional" array. The `.flat()` method is used to "flatten" this structure.

`.flat` says : "I have a list of lists. I just want one single, long list."

`.flat()` creates a new array by pulling out all the items from the sub-arrays. It is non-mutating.

A. Basic Flattening (One Level Deep)

By default, `.flat()` only goes one level deep.

```
const nestedArray = [1, 2, [3, 4]];
const flattened = nestedArray.flat();

console.log(flattened); // [1, 2, 3, 4]
console.log(nestedArray); // [1, 2, [3, 4]] (Original is unchanged)
```

B. Flattening Multiple Levels

You can provide a number as an argument to tell **.flat()** how many levels deep it should go.

```
const deeplyNested = [1, [2, [3, [4]]]];

// Go two levels deep
const flatTwoLevels = deeplyNested.flat(2);
console.log(flatTwoLevels); // [1, 2, 3, [4]]

// To flatten completely, no matter how deep, you can use Infinity.
const completelyFlat = deeplyNested.flat(Infinity);
console.log(completelyFlat); // [1, 2, 3, 4]
```

.flat() also automatically removes empty slots in sparse arrays.

3. Deleting Elements in an Array

There are several ways to "delete" an element, and they have very different results.

Method 1: **.pop()** or **.shift()** (The Cleanest Way)

If you want to remove an element from the beginning or end of an array, these are the best methods. They remove the element and automatically update the **.length**

Method 2: .splice() (The Most Versatile Way)

As we saw before, splice is perfect for removing one or more elements from anywhere in the array. It cleanly removes the elements and re-indexes the array. This is the recommended way to delete from the middle of an array.

```
let items = ["a", "b", "c", "d"];  
// Remove "c" (at index 2)  
  
items.splice(2, 1);  
console.log(items); // ["a", "b", "d"]  
console.log(items.length); // 3
```

Method 3: The delete Operator (The "Bad" Way - Avoid)

You can use the delete operator on an array element, but you should almost never do this.

First Thought: "This seems easy, I'll just delete it."

But it doesn't do what you think. It removes the element's value but leaves an empty, "hole" in its place. It does not update the array's length or re-index the other elements.


```
let letters = ["a", "b", "c"];  
delete letters[1]; // "Delete" the element at index 1  
  
console.log(letters);    // ["a", empty, "c"]  
console.log(letters[1]); // undefined  
console.log(letters.length); // 3 (The length did NOT change!)
```

This creates a sparse array with a "hole" in it, which can cause unexpected behavior in loops and with other array methods.

Conclusion on Deleting:

- To remove from the end, use **.pop()**.
- To remove from the beginning, use **.shift()**.
- To remove from the middle, use **.splice()**.
- Never use the delete operator on an array.

YES! Array is NOT an Array in JS.

Why? Let's Explain

The C++ Array: A Row of Houses

Imagine a new street is built. The government says, "This street will have 10 houses, they will all be identical two-bedroom houses, and they will be built side-by-side in an unbroken row."

1. **Contiguous Memory** : This is the solid foundation. A C++ array (`int arr[10];`) is a single, continuous, unbroken block of memory. Every slot physically exists.
2. **Homogeneous (Identical Houses)** : Every element must be the same type (`int`). This is because every house must be the same size, so the city knows how to find them.
3. **Mathematical Access** : To get to House #7, you don't search. You do a simple calculation: **(Address of House #0) + (7 * size_of_one_house)**. This is why it is blindingly fast.
4. **No Gaps** : You cannot have House #1 and House #7 without also having houses #2, #3, #4, #5, and #6 physically built in between.

This is a true, low-level array. It is a memory layout.

The JavaScript Array: A Wall of P.O. Boxes

Now, imagine a post office. The wall has slots for boxes numbered 0, 1, 2, 3, and so on.

1. **It's an Object (The Wall of Boxes)** : A JavaScript Array is fundamentally an object. The "indices" (0, 1, 2) are not memory offsets; they are property keys, just like "name" or "age" in a regular object. The only difference is that the keys look like numbers.
2. **Can be Sparse (No Gaps Needed)** : This is the most powerful proof. You can have P.O. Box #0 and P.O. Box #1000. You do not need to build the 999 empty boxes in between. The "empty" slots don't actually exist in memory.

Code Proof:

```
let jsArray = [];  
jsArray[0] = "first item";  
jsArray[1000] = "last item";  
  
console.log(jsArray.length); // Outputs: 1001  
console.log(jsArray[500]); // Outputs: undefined
```

A C++ array would have crashed your program here. The JavaScript array is fine. It's just an object that now has two properties, "0" and "1000", and its .length property was automatically updated to **highest_index + 1**. The 999 "empty" slots are just non-existent properties.

3. **Heterogeneous (Different Sized Boxes)** : You can put a tiny letter in Box #0, a large package in Box #1, and a key in Box #2. Each box can hold a different type of thing.¹⁹

Code Proof:

```
let mixedArray = [10, "hello", true, { id: 1 }];
```

This is possible because the array is just an object storing values (or references to values) against keys. A C++ array could never do this because all the "houses" in its contiguous block of memory must be the same size.

4. **Property Lookup (Not Math)** : To get the item at index **1000**, the engine isn't doing a mathematical calculation. It's performing a property lookup on an object, just as if you were asking for **user["name"]**. It's looking for a property with the key **"1000"**.

"If it's just a slow object, why are JavaScript arrays so fast?"

This is where the magic of modern JavaScript engines like V8 comes in. V8 is incredibly smart. It knows that developers want arrays to be fast. So, it has an optimization system:

1. **The Fast Path** : When you create a simple, dense, homogeneous array (like [1, 2, 3, 4]), V8 will say, "Aha! This looks like a real C++ array." Behind the scenes, it will store it in a contiguous block of memory for maximum speed.
2. **The "De-optimization" Trigger** : The moment you do something that breaks the "row of houses" rule, V8 gives up on the optimization and switches back to the slower, more flexible "post office box" (object/hash map) storage. Triggers include:
 - Making the array sparse : **myArray[500] = '...';**
 - Adding mixed types : **myArray.push({});**
 - Deleting an element from the middle : **delete myArray[1];**

Conclusion: Why is a JavaScript Array not a "real" array?

Feature	True Array (C++)	JavaScript Array
Underlying Structure	A contiguous block of memory	A specialized Object with number-like keys
Elements	Must be the same type (homogeneous)	Can be different types (heterogeneous)
Memory Layout	Dense (no gaps allowed)	Can be sparse (can have huge gaps)
Access Method	Fast mathematical offset calculation	Slower property/key lookup (but heavily optimized)

A JavaScript array is not a fundamental memory structure; it is a high-level data structure implemented as an object, which has been given a special `.length` property and an array-like prototype to make it behave like an array for the developer's convenience.